UNIVERSITY OF NOVI SAD
TECHNICAL FACULTY
"MIHAJLO PUPIN"
ZRENJANIN

ITROCONFERENCE7.0

INFORMATION TECHNOLOGY AND EDUCATION DEVELOPMENT

ITROCONFERENCE7.0
INFORMATION TECHNOLOGY AND EDUCATION DEVELOPMENT

PROCEEDINGS

ZRENJANIN, June 2016

UNIVERSITY OF NOVI SAD
TECHNICAL FACULTY ”MIHAJLO PUPIN”
ZRENJANIN
REPUBLIC OF SERBIA

VII INTERNATIONAL CONFERENCE ON

INFORMATION TECHNOLOGY AND DEVELOPMENT OF EDUCATION ITRO 2016

PROCEEDINGS OF PAPERS

VII MEĐUNARODNA KONFERENCIJA

INFORMACIONE TEHNOLOGIJE I RAZVOJ OBRAZOVANJA ITRO 2016

ZBORNIK RADOVA

ZRENJANIN, JUNE 2016

Organiser of the Conference:
University of Novi Sad, Technical faculty „Mihajlo Pupin", Zrenjanin, Republic of Serbia

Publisher:
University of Novi Sad, Technical faculty „Mihajlo Pupin", Djure Djakovica bb, Zrenjanin, Republic of Serbia

For publisher:
Dragica Radosav, Ph. D, Professor, Dean of the Technical faculty „Mihajlo Pupin", Zrenjanin, Republic of Serbia

Editor in chief:
Marjana Pardanjac, Ph. D, Assistant Professor, Technical faculty „Mihajlo Pupin", Zrenjanin, Republic of Serbia

Technical treatment and design:
Ivan Tasic, Ph. D, Professor
Dijana Karuovic, Ph. D, Professor
Vesna Makitan, Ph. D, Assistant Professor
Erika Eleven, M.Sc, Assistant
Dusanka Milanov MSc, Assistant

Lecturer:
Erika Tobolka, Ph. D, Professor

Printed by:
Printing office SAJNOS DOO, Momčila Tapavice 2, Novi Sad, R. of Serbia

Circulation: **50**
ISBN: 978-86-7672-285-3

CIP - Каталогизација у публикацији
Библиотека Матице српске, Нови Сад

37.01:004(082)
37.02(082)

INTERNATIONAL Conference on Information Technology and Development of Education ITRO (7 ; 2016 ; Zrenjanin)
Proceedings of papers / VII International Conference on Information Technology and Development of Education ITRO 2016 = Zbornik radova = VII međunarodna konferencija Informacione tehnologije i razvoj obrazovanja ITRO 2016, Zrenjanin, June 2016. - Zrenjanin : Technical Faculty "Mihajlo Pupin", 2016 (Novi Sad : Sajnos). - VI, 413 str. : ilustr. ; 30 cm

Tekst štampan dvostubačno. - Tiraž 50. - Introduction: str. VI. - Bibliografija uz svaki rad.

ISBN 978-86-7672-285-3

a) Информациона технологија - Образовање - Зборници b) Образовна технологија - Зборници
COBISS.SR-ID 306831623

PARTNERS INTERNATIONAL CONFERENCE

South-West University „Neofit Rilski"
Faculty of Education, Blagoevgrad,
Republic of Bulgaria

SOUTH WEST UNIVERSITY
"NEOFIT RILSKI"

Faculty of Electrical Engineering and Informatics
Department of Computers and Informatics of Kosice
Slovak Republic

University Goce Delcev Stip
Republic of Macedonia

УНИВЕРЗИТЕТ
„ГОЦЕ ДЕЛЧЕВ"
ШТИП

THE SCIENCE COMMITTEE:

With this publication, the CD with all papers from the International Conference on Information Technology and Development of Education, ITRO 2016 is also published.

INTRODUCTION

This Proceedings of papers consists from full papers from the International conference "Information technology and development of education" - ITRO 2016, that was held at the Technical Faculty "Mihajlo Pupin" in Zrenjanin on June 10th 2016.

The International conference on Information technology and development of education has had a goal to contribute to the development of education in Serbia and the Region, as well as, to gather experts from natural and technical sciences' teaching fields.

The expected scientific-skilled analysis of the accomplishment in the field of the contemporary information and communication technologies, as well as analysis of state, needs and tendencies in education all around the world and in our country has been realized.

The authors and the participants of the Conference have dealt with the following thematic areas:

- Theoretical and methodological questions of contemporary pedagogy
- Personalization and learning styles
- Social networks and their influence on education
- Children security and safety on the Internet
- Curriculum of contemporary teaching
- Methodical questions of natural and technical sciences subject teaching
- Lifelong learning and teachers' professional training
- E-learning
- Education management
- Development and influence of IT on teaching
- Information communication infrastructure in teaching process

All submitted papers have been reviewed by at least two independent members of the Science Committee.

There were total of 163 authors that took part at the Conference from 15 countries, 4 continents: 96 from the Republic of Serbia and 67 from foreign countries such as: Macedonia, Bulgaria, Slovakia, Russia, Montenegro, Albania, Hungary, Italy, India, Rumania, Bosnia and Herzegovina, USA, Egypt and Nigeria. They were presented 82 scientific papers; 42 from Serbia and 40 from the above mentioned countries.

The papers presented at the Conference and published in Proceedings can be useful for teachers while learning and teaching in the fields of informatics, technics and other teaching subjects and activities. Contribution to the science and teaching development in this Region and wider has been achieved in this way.

The Organizing Committee of the Conference

CONTENTS

INFORMATION COMMUNICATION INFRASTRUCTURE IN TEACHING PROCES

International Conference on Information Technology and Development of Education – ITRO 2016
June, 2016. Zrenjanin, Republic of Serbia

Benefit of Using Shared Memory in Implementation of Parallel FWT Algorithm with CUDA C on GPUs

D. Bikov*, I. Bouyukliev**, A. Stojanova*
*Faculty of Computer Science "Goce Delcev" University -UGD Stip, Republic of Macedonia
** Institute of Mathematics and Informatics, BAS, Veliko Tarnovo, Republic of Bulgaria
dusan.bikov@ugd.edu.mk, iliyab@math.bas.bg, aleksandra.stojanova@ugd.edu.mk

Abstract - GPUs have different memory hierarchy than CPU and with their proper use, we can achieve effective implementation and improve the performance. In this paper we discuss how to use shared memory on GPUs and how does it affect the implementation and performance. For a more detail clarification of benefit of using shared memory, we take into account parallel algorithm for calculation of the Walsh spectrum on graphics processor unit (GPU) and its parallel implementation in CUDA C. Using shared memory is a good optimization strategy, which gives faster time of execution of the parallel program.

I. Introduction

Challenge of modern computer architecture is building fast computation model and for that data movement must be fast. In addition, is needed lot of memory for big applications and fast computation for big data can be very expensive.

Memory hierarchy must be considered when we write the parallel code. Execution speed relies on exploiting data memory locality. Here we will explain the memory model and how it can be exploited to obtain better performance.

The use of modern graphics processing units (GPUs) has become attractive for scientific computing which is due to its massive parallel processing capability. Modern GPUs are more than very efficient device use for rendering the graphics and accelerate the creation of images. Their highly parallel structure makes them more effective than general-purpose CPU for algorithm where processing of large blocks of data is done in parallel [1] [2]. Compared with multi-core CPUs, new generation GPUs can have much higher computation power and memory bandwidth. Therefore, they are attractive in many application areas. One of the most important application domains is the linear algebra [3] [4].

The purpose of this paper is to assess the performance of the recent, inexpensive and widely used NVIDIA GPUs in performing Walsh Transforms. Our approach for the calculation of the Walsh spectrum is a Fast Walsh Transform and briefly, we will describe the mathematical background below in this paper. First, we will show the basic algorithm which can be easy to implement using the shared memory and we will show how this algorithm can affect the performance.

A. *Overview of this paper*

In Section II we give a brief introduction in GPU Computing model with CUDA. In Section III we present the mathematical background for a Walsh transform. In Section IV we explain the basic Algorithm and Algorithm with shared memory. We summarize the results in Section V and give some conclusions.

II. GPU Computing Model with Cuda

GPUs are designed for efficient execution of thousands of threads in parallel on as many processors as possible at each moment. The computation processes are divided into many simple tasks that can be performed at the same time. This intensive multi-threading allows execution of various tasks on the GPU processors while data is fetched from or stored to the GPU global memory. It also ensures the scalability of the GPU computing model, since processors are abstracted as threads, and support parallel programming model [5].

A. *CPU versus GPU*

A simple way to understand the difference between a CPU and GPU is to compare how they process tasks. A CPU consists of a few cores optimized for sequential serial processing while a GPU has a massively parallel architecture consisting of thousands of smaller, more efficient cores designed for handling multiple tasks simultaneously. This ability of a GPU with hundred and more cores to process thousands of threads can significantly accelerate the software over a CPU.

B. The CUDA programming model and hardware interface

Modern NVIDIA GPUs are powerful computing platform developed for general purpose computing using CUDA (Compute Unified Device Architecture) [6]. It allows programmers to interact directly with the GPU and run programs on them, thus effectively utilizing the advantages of parallelization. Depends of architecture CUDA cores can be organized into SMs (streaming multiprocessor), each having a set of registers, constants and texture caches, and on-chip shared memory as fast as local registers (one cycle latency). At any given cycle, each core executes the same instruction on different data (SIMD), and communication between multiprocessors is performed through global memory. As a programming interface, CUDA C is not a new language, it is a set of C language library functions with GPU specific commands, options and operations [7], and the CUDA-specific nvcc compiler generates the executable for the NVIDIA GPU from a source code.

Data-parallel functions are written in units called kernels which execute over the stream of data by many thread on a device in parallel. Thread is a process that performs series of independent programming instruction and is single instance of the kernel. Creating and destroying of thread barely requires resources (time), from where come that they don't have any significant impact of the performances. Threads are organized into blocks, which are sets of treads that can communicate and synchronize their execution. It can launch maximum 1024 threads (512 threads for older GPU) per block. Block are executed by single SMs, depending on the specific GPU hardware, a SMs can execute multiple blocks simultaneously [5]. Block and thread per block form a grid.

C. Memory hierarchy

It must to consider memory hierarchy while write parallel code. Execution speed relies on exploiting data locality. Lowest level memory is the faster which itself is more expensive and smaller (limited). Registers are the fastest followed by local memory, shared memory and global memory. Every thread has access to his local memory. Data in share memory can share between every thread of the same block. All threads from all kernels can access to global memory. Since blocks execute in an arbitrary order, if one block modifies a data element, no other block should read or write that data element in global memory. Except these types of memory, there are additional memory and variable types. Constant can only read and reading it is almost fast as a reading from register. Texture memory originally, intended primarily for pure graphics applications.

There are several benefit from using shared memory, they can use for operations who requiring communication between threads, it is useful for data re-use, it is alternative for local memory, it reduces use of registers when a variable has same value for all threads.

Memory model shows interaction of the threads and there is a chance thread to read result from computation before other thread to write (compute) or with other words we can obtain incorrect result. Consequently, usually thread synchronization is needed to ensure correct use of the memory. Instruction __syncthreads(); inserts a "barrier" synchronization no thread in a block is allowed to proceed beyond this point until the rest have reached it. Global synchronization of all threads can be performed across separate kernel launches or with Fast Barrier Synchronization [8].

III. LINEAR BOOLEAN FUNCTIONS AND WALSH SPECTRUM

Let $f(x)=u_1x_1 \oplus u_2x_2 \oplus \ldots \oplus u_nx_n$ be a linear Boolean function of n variables. We use the notation $u_1x_1 \oplus u_2x_2 \oplus \ldots \oplus u_nx_n = x^{(\oplus u)}$. The binary n -dimensional vector u uniquely defines $f(x)$ and therefore we denote it by $f^{(\oplus u)}(x)$. The Truth Table of $f^{(\oplus u)}(x)$ has the form

$$\begin{pmatrix} f^{(\oplus u)}(\overline{0}) \\ f^{(\oplus u)}(\overline{1}) \\ \vdots \\ f^{(\oplus u)}(\overline{2^n-1}) \end{pmatrix} = \begin{pmatrix} \overline{0}^{(\oplus u)} \\ \overline{1}^{(\oplus u)} \\ \vdots \\ \overline{(2^n-1)}^{(\oplus u)} \end{pmatrix} = (S_{mat}^{(n)})^{(\oplus u)}$$

The values of the linear functions for $\overline{0},\overline{1},\ldots,\overline{2^n-1}$ form the following matrix:

$$H_n^+ = \begin{pmatrix} \overline{0}^{\oplus\overline{0}} \; \overline{0}^{\oplus\overline{1}} & \cdots & \overline{0}^{\oplus\overline{2^n-1}} \\ \overline{1}^{\oplus\overline{0}} \; \overline{1}^{\oplus\overline{1}} & \cdots & \overline{1}^{\overline{2^n-1}} \\ & \cdots & \\ \overline{(2^n-1)}^{\oplus\overline{0}} \; \overline{(2^n-1)}^{\oplus\overline{1}} & \cdots & \overline{(2^n-1)}^{\oplus\overline{2^n-1}} \end{pmatrix}$$

Hence

$$H_n^+ = \left((S_{mat}^{(n)})^{\oplus \bar{0}},\ (S_{mat}^{(n)})^{\oplus \bar{1}}\ ,\dots,\ (S_{mat}^{(n)})^{\oplus \overline{2^n-1}} \right)$$

$$= \left(\begin{pmatrix} 0S_{mat}^{(n-1)} \\ 1S_{mat}^{(n-1)} \end{pmatrix}^{\oplus \bar{0}} \cdots \begin{pmatrix} 0S_{mat}^{(n-1)} \\ 1S_{mat}^{(n-1)} \end{pmatrix}^{\oplus \overline{2^{n-1}-1}} \cdots \begin{pmatrix} 0S_{mat}^{(n-1)} \\ 1S_{mat}^{(n-1)} \end{pmatrix}^{\oplus \overline{2^n-1}} \right)$$

For the matrix H_n^+ we have

$$\left(\left(0S_{mat}^{(n-1)}\right)^{\oplus \bar{0}} \dots \left(0S_{mat}^{(n-1)}\right)^{\oplus \overline{2^{n-1}-1}} \right) =$$

$$\left(\left(1S_{mat}^{(n-1)}\right)^{\oplus \bar{0}} \dots \left(1S_{mat}^{(n-1)}\right)^{\oplus \overline{2^{n-1}-1}} \right) = H_{n-1}^+,$$

$$\left(\left(1S_{mat}^{(n-1)}\right)^{\oplus \overline{2^{n-1}}} \dots \left(1S_{mat}^{(n-1)}\right)^{\oplus \overline{2^{n-1}}} \right) = \overline{H_{n-1}^+},$$

where the matrix $\overline{H_{n-1}^+}$ is obtained from H_{n-1}^+ after replacing 0 by 1 and 1 by -1. It follows that

$$H_{n-1}^+ = \begin{pmatrix} H_{n-1}^+ & H_{n-1}^+ \\ H_{n-1}^+ & \overline{H_{n-1}^+} \end{pmatrix}, H_1^+ = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix},$$

$$H_2^+ = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix} .$$

It is easy to see that H_n is a symmetric matrix. Its rows (and columns) form n dimensional linear space. In coding theory this space (without zero coordinate) is known as a simplex code. This space together with its coset with representative (11 . . . 1), form the first order Reed-Muller code.

Let $a = (a_1, a_2, ..., a_m)$ be a binary vector. The polarity representation $a^{(p)}$ of a is obtained from a after replacing 0 by 1 and 1 by −1. Consider the scalar product $s = a^{(p)} \cdot b^{(p)}$ over the integers. Let s^- (respectively s^+) be the number of the coordinates, for which $a_j^{(p)} b_j^{(p)} = -1$ (respectively $a_j^{(p)} b_j^{(p)} = +1$). Then $s^- = d(a,b)$ is the number of coordinates with different value for a and b . And s^+ is the number of the coordinates with equal values for a and b . We have that $s = s^+ - s^-$ and $m = s^+ + s^-$ or

$$s^- = (m+s)/2, s^+ = (m+s)/2 .$$

Let us denote by $PTT(f)$ and H the polarity representations of $TT(f)$ and the matrix H^+ . The vector

$$W_f = H \cdot (PPT(f))^t =$$

$$(f^w(\bar{0}), f^w(\bar{1}), ..., f^w(\overline{2^n-1}), W_f = (W_0, ..., W_{2^n-1})$$

is called Walsh spectrum, and the function $f^w(\bar{a})$ defines the Walsh transform. The value W_i determines the distance between the Truth Table of f and the Truth Table of the linear function $x^{\oplus i}$, which is equal to $(2^n - W_i)/2$, and the distance between $TT(f)$ and the Truth Table of the affine function $1 + x^{\oplus i}$ which equal to $(2^n - W_i)/2$.

Matrix vector multiplication $H \cdot (PPT(f))^t$ can be given by a butterfly diagram and a corresponding algorithm, namely Diagram 2 and Algorithm 2 [8]. This algorithm passes all elements of the matrix $S_{mat}^{(n)}$ in n steps column by column starting from the last one. Depending on the value in the i -th row and $(n-j+1)$ -th column of the matrix $S_{mat}^{(n)}$ the algorithm calculates a new values for $W_f[i]$ and $W_f[i+2^i]$. This algorithm entirely depends on the binary representation of the nonnegative integers smaller than 2^n .

Fast Walsh Transform can be implement parallel, by using base concept on Algorithm 2 [9] but with acceptable modification to be suitable for parallel implementation. For our parallel implementation, we use CUDA C and we make several versions where we use various optimization techniques, model and different memory to get better performance and efficiency.

IV. PARALLEL ALGORITHMS

In introduction, we mentioned that we would show basic algorithm for Fast Walsh Transform and then improvements with using of shared memory. These algorithms are implemented in CUDA C.

The basic algorithm (Algorithm 1) is based on the sequential Algorithm 2 [9] but with suitable modification in order to implement it in parallel. Below is shown pseudo code from parallel implementation of the Algorithm 1 of Fast Walsh Transform.

Algorithm 1. Parallel implementation of FWT

Input: The Polarity Truth Table PTT of the Boolean function f, with 2^n entries

Output: The Walsh spectrum W_f of the Boolean function f, with 2^n entries

```
// Allocate memory for device copies and host copy
// Copy inputs to device
// Set grid of block and thread
j ← 1;
r ← 0;
W_f ← PPT
while (j < 2^n ) do
    r ← r + 1;
    fwt_kernel(W_f , temp, r, j) /*Launch kernel*/
    j ← j · 2;
end while
// Copy result back to host
// Cleanup memory
```

fwt_kernel(W_f , temp, r, j), Kernel, Algorithm 1

```
fwt_kernel(W_f , temp, r, j)
i ← thread // Set index of the thread
value ← 0; //Local variable for every thread
if(r−(r/2)∗2==1) then /*(r%2==1) Save intermediate results*/
    if(i_[n−j+1] = 0) then / ∗ if((i&j) == 0) ∗ /
        value ← (W_f[i]+W_f[i+j]);
    end then
    else
        value ← (−W_f[i]+W_f[i–j]);
    end else
  temp[i] ← value;
end then
else
    if(i_[n−j+1]=0) then / ∗ if((i&j) == 0) ∗ /
        value ← (temp[i]+temp[i+j]);
    end then
    else
        value ← (−temp[i] + temp[i − j]);
    end else
  W_f[i]←value;
end else
```

Kernel (**fwt_kernel**) from the Algorithm 1 takes the following as input:

- array W_f where to put Polarity Truth Table PTT of the Boolean function f, with 2^n entries
- array temp with 2^n entries, initialized to 0
- variable r initialized to 1, used for tracking where to put result, prevention of rewriting the memory
- a variable j initialized to 1, used for tracking the step and specifying array index

and return the array W_f with 2^n entries, complete Walsh spectrum W_f of the Boolean function f.

This algorithm passes all elements of the matrix $S_{mat}^{(n)}$ in n steps column by column starting from the last one. Depending on the value in the i-th row and (n−j+1)-th column of the matrix $S_{mat}^{(n)}$ the algorithm calculates a new values for $W_f[i]$ and $W_f[i+2^i]$. This algorithm entirely depends on the binary representation of the nonnegative integers smaller than 2^n [9].

Problem is synchronization of threads from different blocks (global synchronization). We performed global synchronization across separate kernel launches. This process doesn't affect program performance because creating and destroying of thread barely requires resources (time). For one-step of computation, the data is read from memory, compute and write new compute data. Every thread according index and location in the block take two memory elements make computation and result write back in the particular memory location.

Except the problem with synchronization, another problem is type of memory used. Global memory is the slowest memory on GPU. With using of shared memory, we want to reduce the problem.

Algorithm with shared memory we combine with memory pattern and we have two kernels. First kernel use shared memory for calculations until certain steps. Because it can launch maximum 1024 threads per block and data is shared between every thread from same block, with shared memory kernel we can calculate until 10 step FWT algorithm (or less steps for less PTT size, element). Below it shown pseudo code from parallel implementation of the Algorithm 2.

Algorithm 2. Parallel implementation of FWT

Input: The Polarity Truth Table PTT of the Boolean function f, with 2^n entries

Output: The Walsh spectrum W_f of the Boolean function f, with 2^n entries

```
// Allocate memory for device copies and host copy
// Copy inputs to device
// Set grid of block and thread
sizeblok ← size/1024 /* if size > 1024 */
sizethread ← max 1024; W_f ← PPT
/*Parallel function, shared memory, Algorithm 2*/
    fwt_kernel_SM(W_f, temp, sizethread)
    if (size > 1024) then
/*Parallel function, memory pattern, Algorithm 2*/
fwt_kernel_SM_MP(temp,W_fresult,sizeblok)
    end then
// Copy result back to host
// Cleanup memory
```

fwt_kernel_SM(W_f , temp, sizethread), Kernel 1,

```
fwt_kernel_SM(W_f , temp, sizethread)
shared tmpsdata; //declare share memory
tid ← thread // Set index of the thread
laneId ← thread + block // Set index of the grid
value ← W_f[laneId]; //Local variable for every thread, take from W_f
for j = 1 to sizethread do
    tmpsdata[tid] = value;
    __syncthreads();
        if(i[n−j+1] = 0) then / ∗ if((i&j) == 0) ∗ /
            value ← (tmpsdata[tid] + tmpsdata[tid+j]);
        end then
        else
            value ← (−tmpsdata[tid] + tmpsdata[tid−j]);
        end else
    __syncthreads();
    temp[laneId] ← value;
    j ← 2 ∗ j
end for
```

First kernel (**fwt_kernel_SM**) takes the following as input:

- array W_f where to put Polarity Truth Table PTT of the Boolean function f, with 2^n entries
- array temp with 2^n entries, initialized to 0
- variable size thread initialized to x ($0 \leqslant x \leqslant 1024$, $x = 2$ n , if(2 n > 1024) ⇒ x = 1024), used for loop in kernel function

and return the array W_f with 2^n entries, complete Walsh spectrum W_f of the Boolean function f. This algorithm passes all elements of the matrix $S^{(n)}_{mat}$ in n steps column by column starting from the last one.

After start of the kernel, shared memory is declared, and data is written from W_f to shared memory. Every thread takes two elements from the shared memory makes addition or subtraction depending on the binary representation of the nonnegative integers smaller than 2^n, stores result in local variable and at the end of step threads are synchronized. If $2^n > 1024$ there is no risk of rewriting data, because computation is separated per block and there is no interaction between threads from different block during the first kernel.

After finishing the first kernel, second kernel begins and proceeds to next step until the last step (depending of PTT size, element). Second kernel is the same like the first one with the addition of memory pattern. Memory pattern is for rearrange the memory in such a way that memory elements from different blocks are set in order to perform FWT from first step. After a certain number of steps again new rearrange is made to obtain proper arrange of the memory.

fwt_kernel_SM_MP(temp, $W_f$result, sizeblok)

```
fwt_kernel_SM_MP(temp, W_f result, sizeblok)
shared tmpsdata; //declare share memory
tid ← thread // Set index of the thread
laneId ← thread + block // Set index of the grid
/*Memory pattern*/
ji = (laneId − (laneId/sizeblok) ∗ sizeblok) ∗ 1024 + (laneId/sizeblok);
value ← temp[ji]; //Local variable for every thread, take from W_f
for j = 1 to sizethread do
    tmpsdata[tid] = value;
    __syncthreads();
        if(i[n−j+1] = 0) then / ∗ if((i&j) == 0) ∗ /
            value ← (tmpsdata[tid] + tmpsdata[tid + j]);
        end then
        else
            value ← (−tmpsdata[tid] + tmpsdata[tid − j]);
        end else
    __syncthreads();
    W_f result[ji] ← value;
    j ← 2 ∗ j
end for
```

Second kernel takes the following as input:

- array temp with 2^n entries, intermediate result from first kernel
- array W_f result where to put final result of the Boolean function f
- variable sizeblok initialized to x ($0 \leqslant x \leqslant 1024$, $x = 2^n/1024$, if($2^n = 2^{20}$) $\Rightarrow x = 1024$), used in expression of memory pattern;

and return the array W_f result with 2^n entries, complete Walsh spectrum W_f of the Boolean function f. Expression which defines us memory pattern is the following one:

$$ji = (laneId - (laneId/sizeblok) * sizeblok) * 1024 + (laneId/sizeblok);$$

here *laneId* is the global index on the thread and *ji* index of array element in memory (memory pattern index).

At beginning of the kernel we declare shared memory and local variable and after declaration we copy *temp* result from the first kernel into local variable *value* of each thread and as an index we use *ji.* Value from the local variable we write in shared memory array, from where we continue with calculation of FWT. After the calculation, memory pattern is use again to return memory in the correct order and write the result back in $W_f$result global memory array. In figure 1 we show memory movement for Boolean function f, with 2048 entries. For Boolean function f, which have more than 2048 entries, is used the same memory pattern but movement of the data will be a little different in order to perform FWT from first step.

V. Algorithms Evaluation

We use the following computer configuration: Intel i3-3110M [10] with 2.4 GHz and 4 GB of RAM and NVIDIA GeForce GT 740M [11], cards with a total of 384 cores running at 0.9 GHz and a 28.8 GB/sec memory bandwidth. The algorithm is implemented in parallel computing platform and programming model CUDA [6]. We use CUDA Toolkit 7.0 and developed environment is MS Visual Studio 2010. Program is executed in Active solution configuration - Release, Active solution platform - Win32.

Speed up is defined by:

$$S_p = \frac{T_{(1,n)}}{T_{p(n)}}$$

where $T_{(1,n)}$ is the run-time of the fastest known sequential algorithm and $T_{p(n)}$ is the run-time of the parallel algorithm, and n is the size of the input.

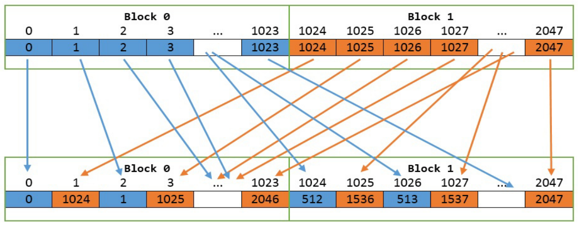

Figure 1. Memory pattern for boolean function f, with 2048 entries

TABLE I.

TABLE I. CPU VS. PARALLEL ALGORITHMS, SPEED UP FOR DIFFRENT NUMBER OF ELEMENT

size	*CPU (ms)*	*Algorithm 1 (ms)*	*Algorithm 2 (ms)*	*Alg. 1 vs Alg. 2*	*CPU vs Alg. 2*
128	0,003	0,024	0,0066	/	/
256	0,007	0,026	0,0066	/	1.060
512	0,015	0,028	0,0069	/	2.272
1024	0,033	0,034	0,0071	/	4.647
2048	0,068	0,039	0,0124	2	5.483
4096	0,145	0,048	0,0147	3,02	9.863
8192	0,308	0,062	0,023	4,967	13.391
16384	0,665	0,096	0,052	6,927	12.788
32768	1,148	0,165	0,13	6,961	8.836
65536	3,116	0,366	0,28	8,513	11.128
131072	6,87	1,561	0,595	4,401	11.546
262144	14,81	3,571	1,207	4,149	12.276

Table I shows time execution for the CPU and GPU algorithms implementations of FWT for different number of elements and speed ups, which appear in the GPU implementation. CPU is faster for small problems and can work faster than couple of threads. For more elements, more threads are used and therefore the computation is faster than in the case of sequential programming. However, there are imitations which depend on several things (the problem, the algorithm, GPUs, the libraries, the model, etc.). Another interesting observation about the Algorithm 2 is intersection of time executions. In some points the memory pattern has higher price (spends more time on memory movement) than shared memory computations. This duplication is due to the memory pattern that is used for reordering the data.

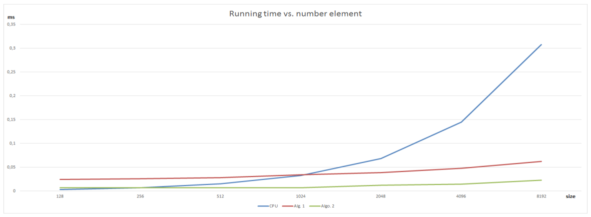

Figure 2. Time for calculating [W_f] CPU vs. Algorithm 1 and Algorithm 2

On Table II and figure 2, we can see the benefit from using the shared memory. Depend on the algorithm 2 design and size of element we obtain variable speed ups. This variation of speed ups also depends on the memory pattern that we use in second kernel, Algorithm 2.

TABLE II. ALGORITHM 1 VS. ALGORITHM 2, SPEED UP

size	*128*	*512*	*2048*	*8192*	*32768*	*131072*
Alg.1 (ms)	0,024	0,028	0,039	0,062	0,165	1,561
Alg.2 (ms)	0,0066	0.0069	0,0124	0,023	0.13	0.595
Speed up	3,63	4.05	3,14	2.69	1.26	2.62

VI. CONCLUSION

In this paper we proposed a performance model for computing Walsh transform with wide use NVIDIA GPU by using popular models in the parallel algorithm community. Here we presented the effect of considering a CPU versus GPU speed. We show how basic algorithm can be improved in order to obtain better performance. Wide used modern GPU has become attractive for scientific computing. This is one of the many examples and here we can see the benefits of using it. Note that here we use low class of GPU.

REFERENCES

[1] 1. D. Gajic, R. Stankovic, GPU Accelerated Computation of Fast Spectral Transforms", Facta universitatis (Nis) Electronics and Energetics 2011 Volume 24, Issue 3, Pages: 483-499, 2011.

[2] 2. D. A. Jamshidi, M. Samadi, S. Mahlke, "D2MA: accelerating coarse-grained data transfer for GPUs " PACT '14 Proceedings of the 23rd international conference on Parallel architectures and compilation Pages 431-442, ISBN: 978-1-4503-2809-8, 2014.

[3] 3. J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. "GPU Computing". Proceedings of the IEEE, 96(5):879–899, May 2008.

[4] 4. P. Maciol and Krzysztof Banas. "Testing tesla architecture for scientific computing: The performance of matrix-vector product." In Computer Science and Information Technology, 2008. IMCSIT 2008. International Multiconference on, pp. 285-291. IEEE, 2008.

[5] NVIDIA, OpenCL Programming Guide for the CUDA Architecture, 2011.

[6] CUDA homepage: http://www.nvidia.com/object/cuda_home_new.html

[7] CUDA Programming Guide: http://docs.nvidia.com/cuda/#axzz3HNpg3SNW.

[8] Shucai Xiao and Wu-chun Feng. Inter-block GPU communication via fast barrier synchronization. IEEE, Page(s):1 - 12, ISSN:1530-2075, 2010.

[9] Iliya Bouyukliev, Dusan Bikov. Applications of the binary representation of integers in algorithms for boolean functions. SMB, 2015.

[10] i3-3110M specifications,http://ark.intel.com/products/65700/Intel-Core-i3-3110M-Processor-3M-Cache-2 40-GHz

[11] NVIDIA GeForce GT 740M specification, http://www.geforce.com/hardware/notebook-gpus/geforce-gt-740m/specifications